AD NO._____
REPORT NO. ATC-8563

RESEARCH REPORT

POINT REACTOR

KINETIC ANALYSIS

DARYL E. NEHER II

RADIATION TEAM
SURVIVABILITY/LETHALITY CORE

U.S. ARMY ABERDEEN TEST CENTER
ABERDEEN PROVING GROUND, MD 21005-5059

DECEMBER 2002

U.S. ARMY DEVELOPMENTAL TEST COMMAND
ABERDEEN PROVING GROUND, MD 21005-5055

DISPOSITION INSTRUCTIONS

Destroy this document when no longer needed. Do not return to the originator.

The use of trade names in this document does not constitute an official endorsement or approval of the use of such commercial hardware or software. This document may not be cited for purposes of advertisement.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
December 2002	Final	

4. TITLE AND SUBTITLE
RESEARCH REPORT POINT REACTOR KINETICS ANALYSIS

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Neher II, Daryl E.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Commander
U.S. Army Aberdeen Test Center
ATTN: CSTE-DTC-AT-SL-R
Aberdeen Proving Ground, MD 21005-5059

8. PERFORMING ORGANIZATION REPORT NUMBER
ATC-8653

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Same as Item 8

12. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
A computer code was written using a point reactor kinetics model Program results are compared to previous theoretical and APRF empirical pulse data. The program is used to determine temperature transients for different scram failures. The pulse-less tail mode of operation is discussed.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(Include area code)*
Unclassified	Unclassified	Unclassified	SAR		

TABLE OF CONTENTS

APPENDIXES

(Page ii Blank)

1. INTRODUCTION

This report describes the construction of a computer program, POINTRX, to model the behavior of the Army Pulse Radiation Facility (APRF) pulse research reactor. Parameters such as power, reactivity, and temperature have been calculated as a function of time. The computer model was created so that all significant variables can be input into the code; therefore, it is adaptable for analysis of a variety of nuclear reactor power excursions. The program may be used to conduct a safety analysis of the reactor.

One mode of operation discussed in this report is the pulse-less tail operation. Other facilities with fast burst reactors have conducted small pulse operations where the scramming mechanism is delayed after the pulse, but the purpose of a pulse-less tail operation is to provide a high power level for a short duration. After a pulse, the reactor power will level or plateau at a certain power level, independent of the reactivity insertion. When the reactivity insertion is exactly prompt, critical reactor power rises quickly and levels at the same power of the plateau with no power spike. This type of operation is called a pulse-less tail operation.

2. THEORY AND PROGRAM

The kinetics equations for a point-reactor model are as follows:

$$\frac{dn}{dt} = \frac{\rho - \beta}{\ell} n + \Sigma_i \lambda_i C_i \quad \text{and} \quad \frac{dC_i}{dt} = \frac{\beta_i}{\ell} n - \lambda_i C_i$$

where: n is reactor power, ρ is the reactivity, β_i is the delayed neutron fraction of delayed neutron precursor i, λ_i is the delayed neutron decay constant for precursor i, ℓ is the neutron generation time, and C_i is the delayed neutron precursor population. Note that, the sum of all six β_i is β. Slow transients in a fast reactor requires the solution of systems of equations containing very short time constants. Using the integral form of these equations will allow a numerical solution in which the computer code can control the time step by many orders of magnitude and maintain numeric stability. The integrals are evaluated analytically using the assumption that power follows the form: $n(t)=n_0 e^{At}$. Therefore, the only numerical approximation is the assumption that A is constant in the above equation throughout the time interval.

Substitute the integral form for C from the second equation into the first equation, and integrate to obtain an equation for power:

$$n(t) = n_0 + \frac{1}{\ell}\int_{t_0}^{t} \rho(t')n(t')dt' + \Sigma_i \lambda_i C_{i_0} \int_{t_0}^{t} e^{-\lambda_i (t'-t_0)} dt' - \Sigma_i \frac{\beta_i}{\ell} \int_{t_0}^{t} n(t') e^{-\lambda_i (t-t')} dt'$$

The equation for reactivity has the form:

$$\frac{d\rho}{dt} = \alpha n(t)$$

$$\text{so, } \rho(t) = \rho_0 + \alpha \int_{t_0}^{t} n(t')dt'$$

and with the assumption:

$$n(t)=n_0e^{At},\ \rho(t) = \rho_0 + \frac{\alpha n_0}{A}\left(e^{A(t-t_0)} - 1\right)$$

where: α, in units of *dk*/kW-sec, is the negative reactivity coefficient, which in this case includes the conversion of reactor power into heat generation. Substituting this result and our assumption into the power equation, we can integrate to get an equation that can be solved numerically:

$$n(t) = n_o + \frac{\rho_0 n_0}{\ell A}\left(e^{Ah} - 1\right) + \frac{\alpha n_0^{\,2}}{\ell A^2}\left(\frac{1}{2}(e^{2Ah} + 1) - e^{Ah}\right) - \Sigma_i C_{i_0}(e^{-\lambda_i h} - 1) - \Sigma_i \frac{\beta_i n_0 e^{-\lambda_i h}}{\ell(A + \lambda_i)}\left(e^{(A+\lambda_i)h} - 1\right)$$

where: h is $t-t_0$.

A computer code, POINTRX, was written to solve the equation for A for a small time increment, h. After A has been determined, reactor power and reactivity are easily calculated. These values are then used as the basis for the next time increment. The complete derivation of these equations and the formulas used in the code are found in Appendix A. The source code for POINTRX, with a sample input file, is attached as Appendix B.

The reactivity coefficient of -0.3 cents/°C was used in the analysis. This negative reactivity coefficient was obtained from the APR Core Design Summary, L. Goldstein (1966). The axial and radial coefficients are combined and were calculated with two-dimensional transport theory using the S_4 approximation with six neutron energy groups. Additionally, in order to calculate α in the code, it is necessary to determine the heat capacity of the core in units of °C/kW-sec. From previous high-power, steady-state operations at 5 and 10 kW, where cooling was not used, the indicated reactor temperature rise was divided by the integrated power for that interval and determined to be 0.04683 °C/kW-sec. The graphs used for determination of temperature rise per kW-sec are included in Appendix C.

Since these data were obtained from in-core thermocouple No. 7, the temperature data from the program should be indicative of thermocouple No. 7 data. However, the APRF technical specifications point out that the peak to measured ratio is much smaller for steady-state operations than pulse operations. Since the 5 and 10 kW operations were relatively short in duration, the program is expected to produce data that are only slightly less than measured data. Furthermore, it should be pointed out that the program has no method for removing heat; thus, heat in the core is always accumulating. This is a reasonable approximation during the pulse as an insignificant amount of heat would have dissipated in the short time elapsed; however, when investigating temperature rise over long time intervals, the results will be conservative.

A value of 9.7 ns for the prompt neutron lifetime was reported by J. T. Mihalczo (1969) and calculated from measurements using both Rossi-α and pulsed neutron techniques. APRF Memorandum for Record 78-68 uses a value of 11.25 ns but does not cite a reference and BRL Contract Report No. 82 also uses a value of 11.25 ns and cites the Mihalczo report, even though this is incorrect. Furthermore, a value of 11.26 ns was calculated using a simple APRF reactor model, by Monte Carlo method, using the MCNP4C code. From calculations using POINTRX, it was discovered that the neutron generation time only affects the prompt period and pulse width. Changing the generation time had no effect on the temperature rise or integrated power, which is of more concern for safety analysis aspects; thus, the neutron generation value of 10 ns was used in the calculations.

The safety block drop time of 200 ms was obtained from safety block drop test records. This test uses a digital oscilloscope to record the time that the scram signal is received from the scram switch and to signal from the safety block out-switch on the reactor package. The 200 ms is the time the safety block takes to be completely out of the core. From previous pulse records in automatic mode, the timer stopped an average of 50 ms after the neutron generator fired. This is the time when the safety block starts to move since the clock stops when the safety block magnet is no longer engaged. Thus, the safety block magnet collapses in 50 ms and the travel time for the safety block is 150 ms. For simplicity of this study, the safety block was removed linearly, \$1 every 10 ms up to 150 ms for a total of \$15. The actual removal of the safety block would start slow and proceed quicker with time; therefore, this study uses a conservative reactivity worth of \$15 for the safety block and which is reported as \$20 by the APRF Safety Analysis Report.

An analysis is performed of the circumstance where the safety block fails to fall after a pulse but the pulse rod does retract. The safety block drop test method was used in analyzing the timing of the pulse rod. The time from the scram signal to the movement of the pulse rod was measured to be 160 ms. The drop time for the pulse rod to be completely out of the core was measured to be 360 ms. For this analysis, the pulse rod is removed in the same manner as the safety block, \$0.0353 every 30 ms up to 360 ms. Normally, the rise time in a pulse is fast enough that the safety channels will trip a scram within hundreds of microseconds of the start of the pulse; therefore, time zero is sufficient to start the time delay of the movement of the safety block or pulse rod. The drop test and pulse rod data sheets are included as Appendix D.

3. RESULTS AND ANALYSIS

The program calculates a number for A in the code; this number is the inverse of the reactor period, commonly called alpha, but, this is not the same alpha in the reactivity equation. Figure 1 shows the calculated alpha versus reactivity insertion. The points are measured data and the spread is due to instrument error and changes in reactor behavior. By space independent neutron kinetic theory, the slope of the linear relation between the reciprocal prompt reactor period and the core reactivity above prompt critical is the prompt neutron decay constant, or neutron generation time at delayed critical. If we drew a line through the data points (dashed line below) the inverse of this slope yields a prompt neutron lifetime of 11.26 ns; thus, the 10 ns neutron generation time used is a good approximation. However, while the line of calculated data passes very close to zero, the measured data do not seem to pass through the same point.

When measuring the reactivity of the pulse rod prior to a pulse, a mini-pulse is performed inserting approximately 92 cents above delayed critical. Using the Inhour curve, the dynamic reactivity worth of the pulse rod is determined. The dynamic and static worth of the pulse rod are different by a few cents, and thus, there must be a small difference between the pulse rod dynamic worth at 92 cents and above prompt critical. Therefore, the difference of the juxtaposition of the calculated versus measured data, which appears to be between 1 and 2 cents, is the difference in the measured versus true worth of the reactivity insertion.

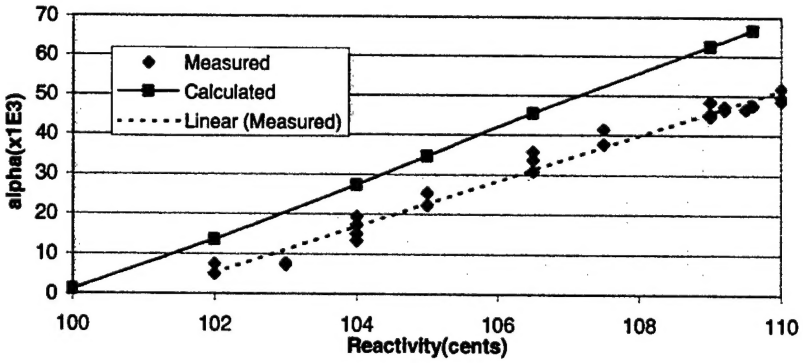


Figure 1. Comparison of measured and calculated alpha.

The program was used to compute the reactor periods for various reactivity insertions. Figure 2 shows the relationship between reactor period and reactivity with the prompt neutron lifetime of 10 ns and 1 ms, with all other variables remaining the same. Compare this to the Inhour curve (fig. 3) from Nuclear Reactor Engineering (Glasstone and Sesonske 1981). The prompt neutron lifetime of 1 ms would be a lifetime associated with a thermal reactor. Since the model is that of a point reactor, relative size is not a factor; thus, the program is also applicable to thermal reactor power excursions with the correct input parameters.

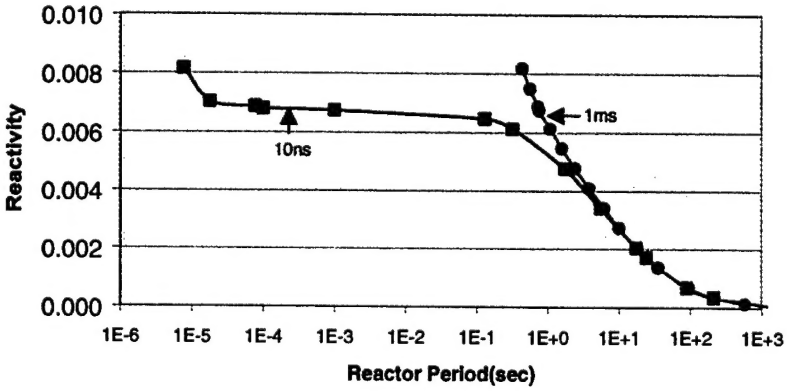


Figure 2. Prompt neutron lifetime comparison.

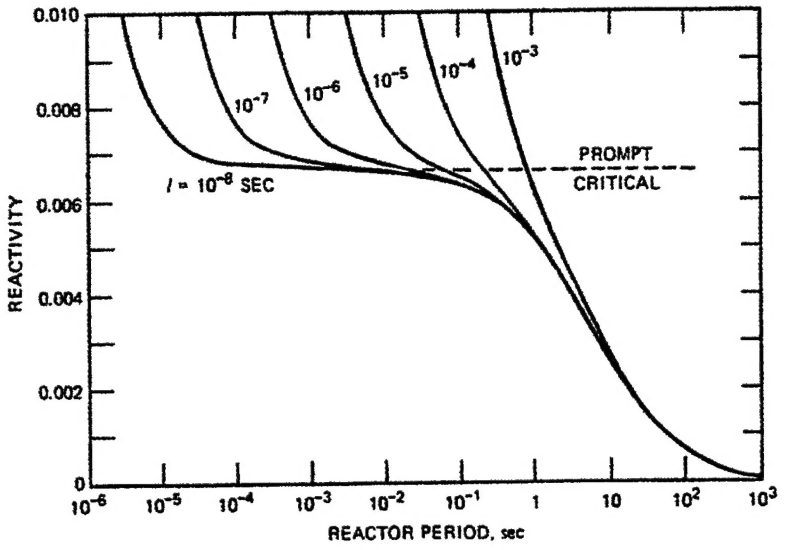


Figure 3. Relationship between reactor period and reactivity for various prompt lifetimes.

Figure 4 shows the shape of a set of pulses calculated by the program on a log/log plot. This shape is similar to the reactor gamma dose rate profile data obtained from photo diode measurements as presented in Figure 5. Note that, after the pulse, the reactor power will plateau until the scram occurs and this plateau region is a significant source of integrated power; thus altering the timing of the scram could change the integrated power by a significant percentage.

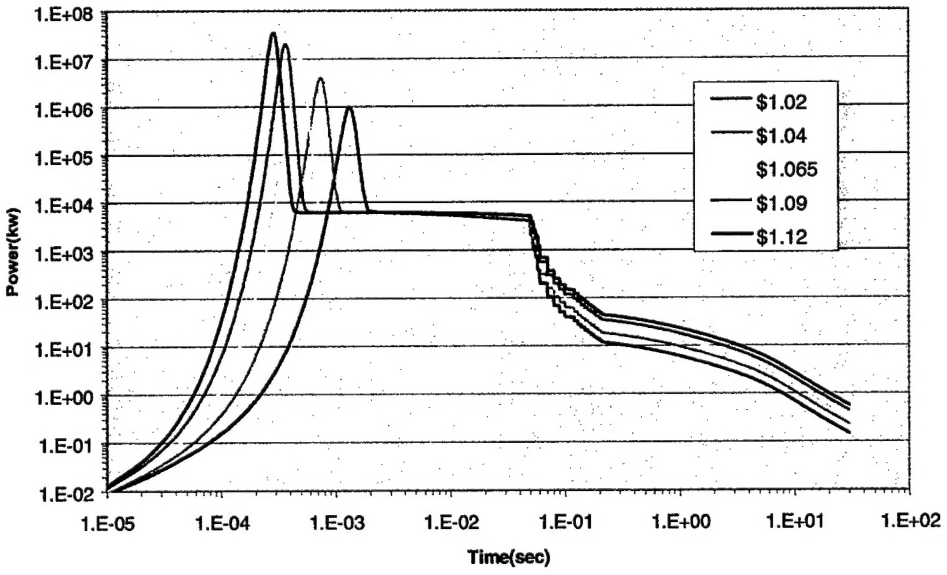


Figure 4. Pulse profiles.

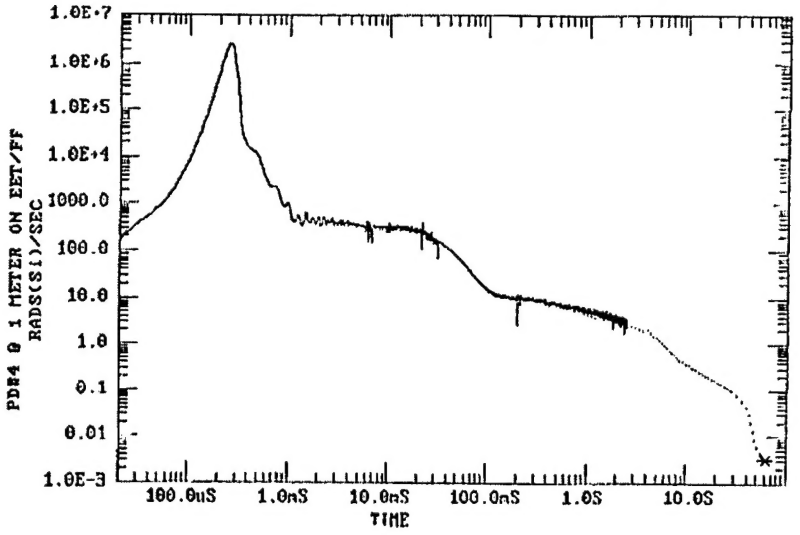


Figure 5. 9.6×10^{16} fission pulse/\$1.088 reactivity insertion.

A comparison of calculated versus measured pulse parameters for various reactivity insertions are presented in Table 1. The data presented here have been calculated out to 30 seconds. The data indicate that pulse-width-at-half-maximum (PWHM) can be predicted with very high accuracy for large pulses, but not for smaller pulses. The differences in the calculated prompt periods and the measured prompt periods are probably an artifact of the method used in the measurement of the dynamic worth of the pulse rod. As mentioned previously, there is a difference of 1 to 2 cents in the true worth of the pulse rod at prompt critical to the dynamic worth measured in a mini-pulse.

TABLE 1. CALCULATED AND MEASURED PULSE PARAMETERS

Reactivity	Integrated Power		Period, μsec		PWHM, μsec		Temperature Change, °C	
	Measured	Calculated	Measured	Calculated	Measured	Calculated	Measured	Calculated
102.0	7.0	9.3	135.0	74	600	278	25	26
104.0	14.5	14.7	73.0	44	175	129	48	42
105.0	19.0	17.5	48.0	29	125	109	66	49
106.5	28.0	21.5	32.0	22	80	82	93	60
109.0	54.0	28.0	21.5	16	56	56	175	79
109.6	62.0	29.5	21.0	15	55	55	210	83

Table 2 shows a comparison of calculated values to the theoretical values of Wimett (1960). For this comparison the calculated values have been converted into the units in Wimett's report. Furthermore, Wimett only calculated the fission yield under the spike; thus the total fission yield is not presented. As expected, the program's results are very close to theoretical values.

TABLE 2. CALCULATED AND THEORETICAL PULSE PARAMETERS

Reactivity	Peak Power, $/sec		Fission Yield, $		PWHM, μsec	
	Wimett	POINTRX	Wimett	POINTRX	Wimett	POINTRX
102.0	135	126	0.04	0.041	261.0	278
104.0	541	500	0.08	0.079	130.0	129
106.5	1413	1370	0.13	0.129	81.0	82
109.0	2813	2640	0.18	0.179	56.3	56

Figures 6 and 7 show a departure of the measured results from calculated results at approximately $1.05 insertion. This is expected at some point because the model does not account for hydrodynamic effects. For large insertions of reactivity, the period becomes increasingly small, as seen in Figure 4. Eventually, the period will be much less than the time required for pressure waves generated in the core to reach the surface. Thus, the core will not expand as quick as the pulse is occurring; therefore, the assumption of a constant reactivity feedback, -0.3cents/°C, will not be accurate.

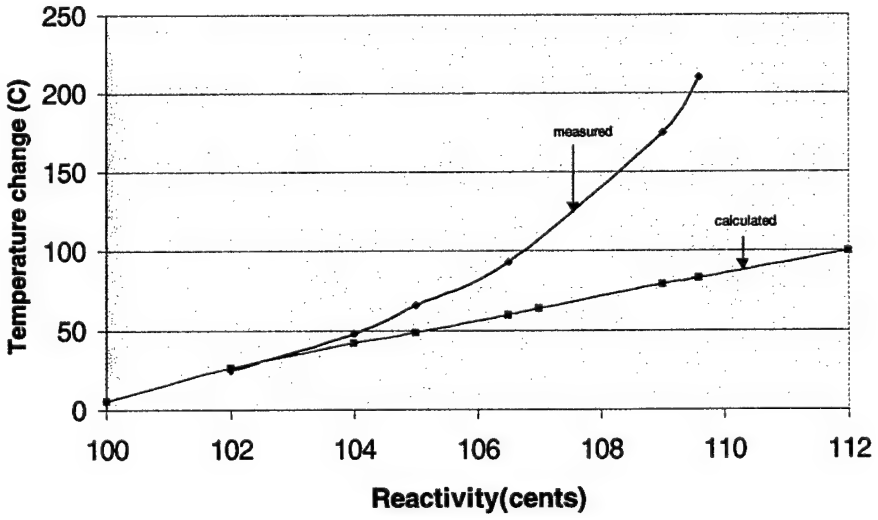


Figure 6. Temperature change versus reactivity insertion.

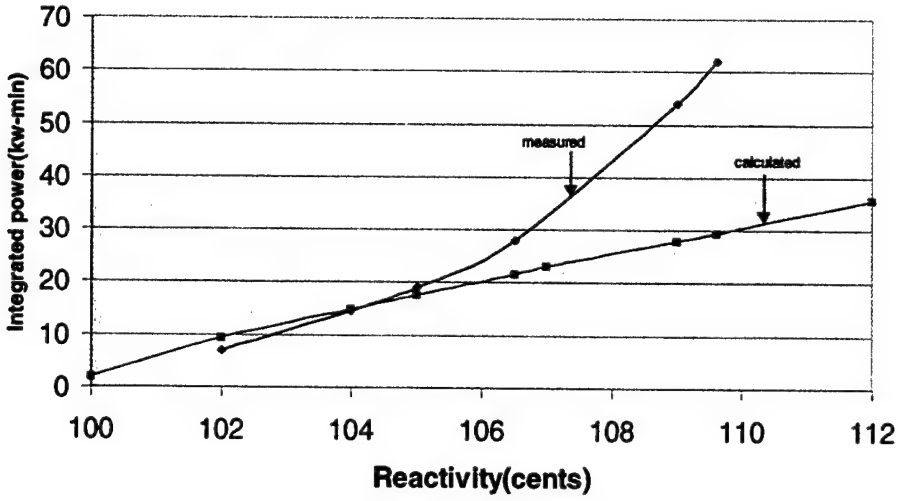


Figure 7. Integrated power versus reactivity insertion.

Even though the program results will deviate from measurements at \$1.05 insertions, this does not necessarily preclude the use of the program for safety analysis. Since the program produces linear relations for temperature and integrated power, the input parameters can be adjusted such that a certain reactivity insertion will calculate equivalent integrated power and temperature results for a \$1.09 insertion. Since the reactivity feedback changes during large pulses, this leads to a change in heat capacity. Using the measured data from Table 1, an effective heat capacity may be obtained by dividing temperature change by integrated power and averaging the results. This gives a heat capacity of 0.05583 °C/kW-sec. This method of deriving the heat capacity may be more accurate than the previously stated method using high power steady-state operations since heat will have less time to migrate during a pulse. Table 3 lists different measured reactivity insertions correlated to a computed reactivity insertion using the new value for heat capacity. Using this value, and a reactivity insertion of \$1.23, which would be a very large pulse in reality, the program will calculate a temperature change and integrated power equivalent to a 10^{17} fission pulse.

TABLE 3. REACTIVITY INSERTION COMPARISON

Reactivity Insertion		ΔT	Calculated ΔT at 120 Seconds		
Measured	POINTRX	Measured	Normal	No Scram	PR Scram
102.0	102	25	26	664	164
104.0	105	48	49	684	217
106.5	111	93	93	725	301
109.0	123	175	180	806	435

Figures 8 and 9 compare the power and temperature excursions of a 10^{17} fission pulse, with a normal scram at 10 kW, to pulses with total and partial scram failures. In the case where no scram occurs there is no significant drop in power, but instead a more gradual decline in power due to the negative reactivity provided by temperature (fig. 10). As seen in Table 3, this is not enough to prevent exceeding the safety limit of 650 °C. Using the peak to measured multiplication factor of 1.43, from the APRF Technical Specifications to determine peak core temperature, the safety limit will be reached in 3.2 seconds and core damage will occur at 60 seconds without any operator intervention.

However, if the only scramming mechanism is the pulse rod, the temperature safety limit will be exceeded in 90 seconds, but no core damage will occur even without operator intervention. This would give the reactor operator sufficient time to take action, such as withdrawing rods to remove more reactivity, thus reducing peak temperatures. Furthermore, these results are considered to be conservative since the heat capacity used to compute alpha in the program is a constant. The heat capacity for U-10 Mo increases as temperature increases; thus, more heat is required to raise the temperature as temperature rises. For small temperature changes this difference is negligible and has been neglected; however, it is more significant with large temperature changes.

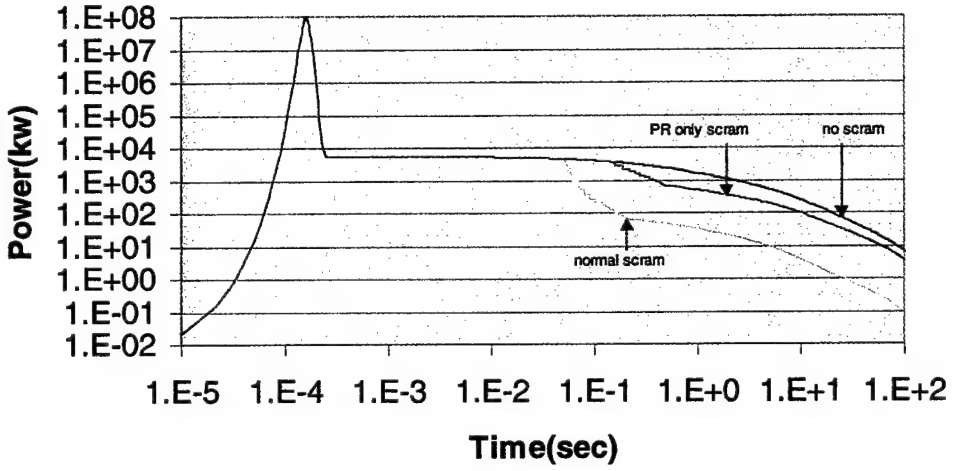


Figure 8. 10^{17} fission pulse power profile.

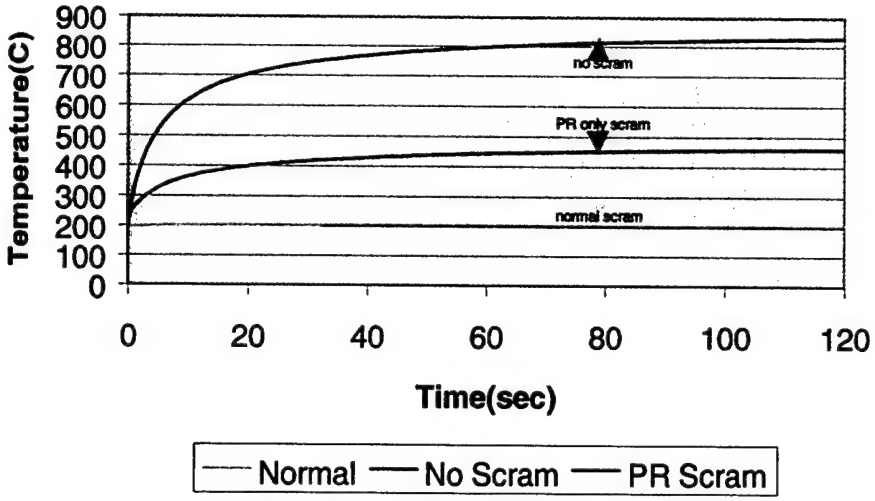


Figure 9. 10^{17} fission pulse temperature excursion.

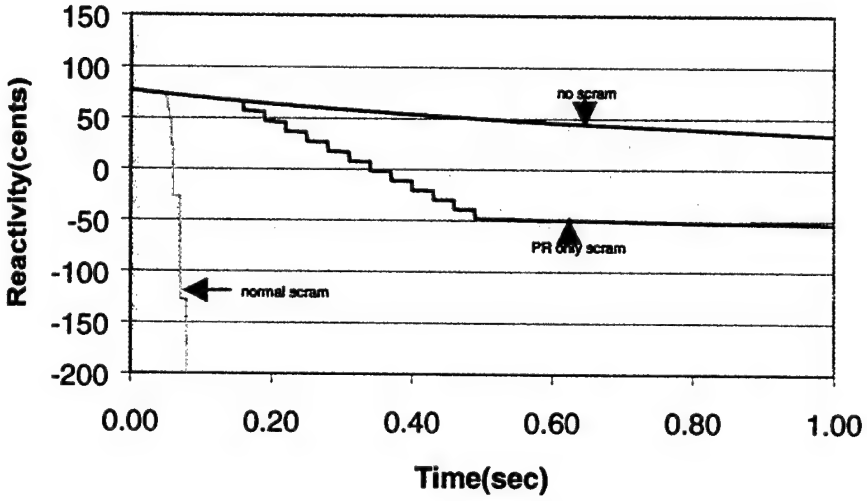


Figure 10. 10^{17} fission pulse reactivity profile.

4. PULSE-LESS TAIL OPERATIONS

It may be possible to take advantage of the plateau power that occurs after a pulse to operate the reactor at very high power levels for short duration. Figure 11 represents the power profiles of reactivity insertions near prompt critical with the scram delayed until 1 second after initiation of the pulse. When the reactivity insertion is exactly prompt critical, there is no power spike, but the power plateau is approximately the same as higher reactivity insertions. Results of the calculated temperatures after 30 seconds for these reactivity insertions with delayed scrams at 1, 5, and 10 seconds are listed in Table 4. The temperatures are initially 25 °C, which is the normal for reactor operations prior to initiating a pulse. Notice that for a normal scram, set point is 10 kW, the pulse below prompt critical has no temperature change. This is comparable to a mini-pulse operation where no temperature change is observable.

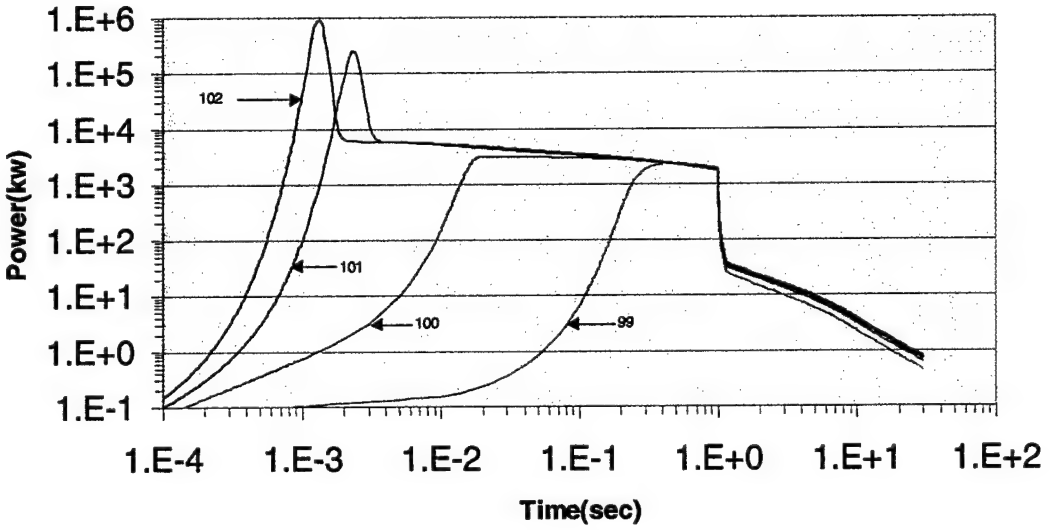


Figure 11. Pulse profile-scram at 1 second.

TABLE 4. CALCULATED TEMPERATURES WITH VARIOUS SCRAMS

Reactivity	Plateau Power	Temperature at 30 Seconds			
		10 kW-Scram	Scram-1 sec	Scram-5 sec	Scram-10 sec
99	2.3 MW	25	106	319	432
100	3 MW	30	138	336	444
101	4 MW	43	152	347	454
102	6 MW	51	163	357	462

By the APRF technical specifications, it is desirable to maintain core temperatures below 350 °C during steady-state operations, although exceptions can be made by the Test Planning Committee to allow operations up to 650 °C. However, any pulse-less tail operation is much faster than the response time of the temperature indication system; therefore, it is not certain whether it would be considered a pulse or a steady-state operation.

From Table 4, the peak fuel temperature change for any of the above pulses where the scram occurs at 1 second would not reach 350 °C. Note again that the temperatures calculated would be indicated temperatures; thus, using the conservative 1.43 peak to measured ratio, the safety limit of 650 °C would not be exceeded if the reactivity insertion is at or below 101 cents and the scram occurred at 10 seconds or less. Furthermore, a pulse-less tail operation is similar to a very wide pulse, in that the thermal stresses generated would be much less than the thermal stress generated in a high yield pulse, which the 650 °C safety limit has been established to prevent.

Table 5 provides calculated temperatures for the same set of insertions with the scram occurring at 5 and 10 seconds; however, for these insertions the safety block is assumed to fail to drop and the pulse rod is the only scramming mechanism. By this table, no permanent damage will occur if the pulse rod drops, and if the pulse rod drops at 5 seconds, the safety limit of 650 °C will not be exceeded.

TABLE 5. CALCULTED TEMPERATURES - PULSE ROD ONLY SCRAM

Temperature at 30 Seconds		
Reactivity	5 sec	10 sec
99	414	491
100	430	503
101	441	512
102	451	521

Figure 12 shows the power level for a 100-cent insertion plotted on a linear scale with a scram occurring at 10 seconds. Figure 13 shows the temperature excursion for the same insertion with a normal scram and a scram failure where the safety block fails to come out, but the pulse rod does come out. Before the scram occurs, the only change to reactivity is the negative temperature feedback. When a normal scram occurs at 10 seconds power drops dramatically and temperature ceases to increase; however, while the small worth of the pulse rod still provides a drop in power, the temperature will continue to rise slowly.

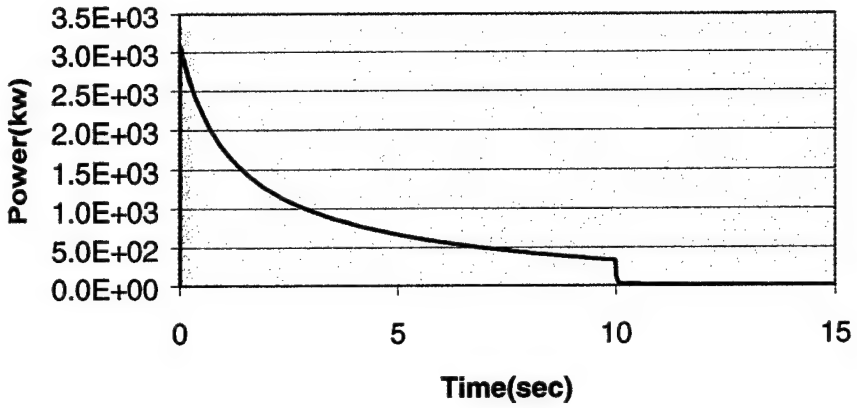


Figure 12. 100 cent insertion - scram at 10 seconds.

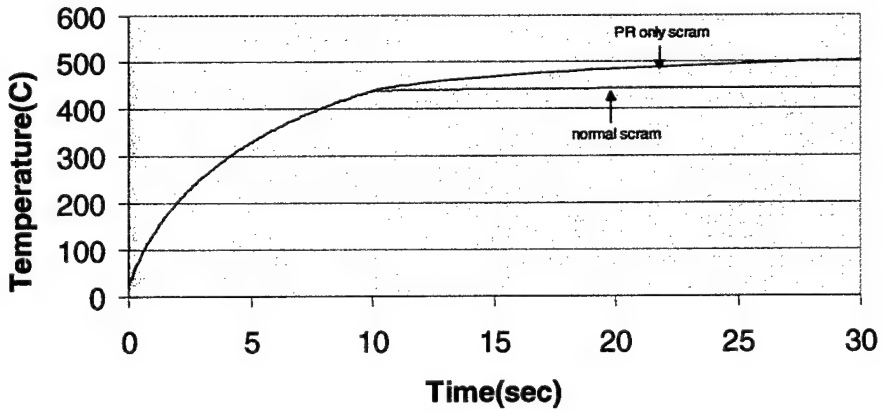


Figure 13. 100 cent insertion - scram at 10 seconds.

5. CONCLUSION

A computer code was written using a point kinetics reactor model to investigate the behavior of various parameters of the APRF pulse research reactor during pulse operations. The calculations have been compared to previous theoretical and APRF empirical data to validate the code. Agreement is good; however, large pulses need to be adjusted because the model does not account for hydrodynamic effects. Results have shown that if the reactor failed to scram after a pulse of 10^{17} fissions, core damage would occur in 60 seconds unless the reactor operator was able to take action. However, if the pulse rod came out of the core, even though the safety block failed to drop, no permanent damage would occur and the operator would have 90 seconds to take action to prevent exceeding a safety limit.

The pulse-less tail operation was investigated for delayed scrams at 1, 5, and 10 seconds. This revealed that for reactivity insertions exactly at prompt critical, no pulse occurs; however, power rises to approximately 3 MW and steadily decreases until the scram occurs. The temperature rise for these operations with normal scrams occurring one second after initiation are well below normal operating temperatures. Furthermore, a safety limit would not be exceeded for an operation up to a 101-cent insertion with a scram delayed up to 10 seconds. It is not obvious at this time whether this operation should be classified as a steady-state or pulse operation.

6. REFERENCES

G. Breidenbach, APR Core Design Summary, United Nuclear Corporation SPAS 66-14, July 1966

A. H. Kazi, H. A. Kurstedt, and V. E. Gazzillo, Preliminary Analysis of the Effect of Youngs Modulus on Fast Pulse Reactor Behavior, Memo for Record 78-68, August 1968.

J. T. Mihalczo, Static and Dynamic Measurements with the Army Pulse Radiation Facility Reactor, ORNL-TM-2330, June 1969.

H. A. Kurstedt, D. E. Glasgow, and T. E. Tipton, Analysis and Monitoring of a Fast Pulse Reactor Core, BRL contract report number 82, August 1970.

H. Kazi, Army Pulse Radiation Facility Reactor Core III Startup Test Summary Report, March 1971.

D. L. Hetrick, Dynamics of Nuclear Reactors, University of Chicago Press, 1971.

S. Glasstone, and A. Sesonske, Nuclear Reactor Engineering third edition, Krieger Publishing Company, 1981.

Technical Specifications for the Army Pulse Radiation Facility, APRF Report 97-6, August 1997.

APPENDIX A DERIVATION OF FORMULAS IN PROGRAM

The point reactor model kinetics equations are:

$$\frac{dn}{dt} = \frac{\rho - \beta}{\ell} n + \Sigma_i \lambda_i C_i \quad (1) \quad \text{and} \quad \frac{dC_i}{dt} = \frac{\beta_i}{\ell} n - \lambda_i C_i \qquad (2)$$

Where the following variables are defined:

N	neutron population
β_i	delayed neutron fraction
λ_i	delayed neutron decay constant
ρ	reactivity
ℓ	prompt neutron lifetime
C_i	delayed neutron precursor population

The precursor equation (2) may be integrated as:

$$C_i(t) = e^{-\lambda_i t}\left[C_{i_o} + \frac{\beta_i}{\ell}\int_0^t n(t')e^{\lambda_i t'}dt'\right] \quad \text{or} \quad C_i(t) = C_{i_0}e^{-\lambda_i(t-t_o)} + \frac{\beta_i}{\ell}\int_{t_0}^t n(t')\,e^{-\lambda_i(t-t')}dt' \qquad (3)$$

To show how equation (3) is derived, assume $C_i(t) = f(t)e^{-\lambda_i(t-t_0)}$, where f(t) is an arbitrary function; thus,

$$\frac{dC_i}{dt} = \frac{df}{dt}e^{-\lambda_i(t-t_0)} + f(t)(-\lambda_i e^{-\lambda_i(t-t_0)})$$

Substituting these equations into equation (2)

$$\frac{df}{dt}e^{-\lambda_i(t-t_0)} + f(t) - \lambda_i e^{-\lambda_i(t-t_0)} = \frac{\beta_i}{\ell}n - \lambda_i f(t)e^{-\lambda_i(t-t_0)}$$

$$\frac{df}{dt}e^{-\lambda_i(t-t_0)} = \frac{\beta_i}{\ell}n$$

$$\frac{df}{dt} = \frac{\beta_i}{\ell}n(t)e^{\lambda_i(t-t_0)}$$

Integrate to get

$$f(t) = \int_{t_0}^t \frac{\beta_i}{\ell}n(t')e^{\lambda_i(t'-t_0)}dt' + \text{const}$$

Substitute f(t) into the assumption to get equation (3).

$$C_i(t) = C_{i_0} e^{-\lambda_i (t-t_0)} + \frac{\beta_i}{\ell} \int_{t_0}^{t} n(t') e^{\lambda_i (t'-t_0)} dt'$$

Substitute equation (3) into equation (1) to get:

$$\frac{dn}{dt} = \frac{\rho - \beta}{\ell} n(t) + \Sigma_i \lambda_i \, C_{i_0} e^{-\lambda_i (t-t_0)} + \Sigma_i \frac{\lambda_i \beta_i}{\ell} \int_{t_0}^{t} n(t') e^{-\lambda_i (t-t')} dt'$$

Integrate to get

$$n(t) = n_0 + \int_{t_0}^{t} \frac{\rho(t') - \beta}{\ell} n(t') dt' + \Sigma_i \lambda_i C_{i_0} \int_{t_0}^{t} e^{-\lambda_i (t'-t_0)} dt' + \frac{\Sigma_i \lambda_i \beta_i}{\ell} \int_{t_0}^{t} dt' \int_{t_0}^{t'} e^{-\lambda_i (t'-t'')} n(t'')$$

Evaluate double integral using integration by parts

$$\int u dv = uv - \int v du$$

$$u = \int_{t_0}^{t'} n(t'') e^{\lambda_i t''} dt'' \qquad du = n(t') e^{\lambda_i t'} dt'$$

$$dv = e^{-\lambda_i t'} dt' \qquad v = \frac{-1}{\lambda_i} e^{-\lambda_i t'}$$

$$\int_{t_0}^{t} \underbrace{e^{-\lambda_i t'} dt'}_{dv} \int_{t_0}^{t'} \underbrace{n(t'') e^{\lambda_i t''} dt''}_{u} = \underbrace{\int_{t_0}^{t'} n(t'') e^{\lambda_i t''} dt''}_{u} \underbrace{\left[-\frac{1}{\lambda_i} e^{-\lambda_i t'} \right]}_{v} - \int_{t_0}^{t} \underbrace{-\frac{1}{\lambda_i} e^{-\lambda_i t'}}_{v} \underbrace{n(t') e^{\lambda_i t} dt'}_{du}$$

$$= \frac{-1}{\lambda_i} \int_{t_0}^{t'} n(t'') e^{-\lambda_i (t'-t'')} + \frac{1}{\lambda_i} \int_{t_0}^{t} n(t') dt'$$

$$t'' \to t' \quad t' \to t$$

$$= -\frac{1}{\lambda_i} \int_{t_0}^{t} n(t') e^{-\lambda (t-t')} dt' + \frac{1}{\lambda_i} \int_{t_0}^{t} n(t') dt'$$

Substitute this back into power equation.

$$n(t) = n_0 + \int_{t_0}^{t} \frac{\rho(t') - \beta}{\ell} n(t') dt' + \Sigma_i \lambda_i C_{i_0} \int_{t_0}^{t} e^{-\lambda_i (t'-t_0)} dt' + \frac{\Sigma_i \beta_i}{\ell} \int_{t_0}^{t} n(t') dt' - \frac{\Sigma_i \beta_i}{\ell} \int_{t_0}^{t} n(t') e^{-\lambda_i (t-t')} dt'$$

Since $\Sigma_i\beta_i = \beta$

$$n(t) = n_0 + \frac{1}{\ell}\int_{t_0}^{t} \rho(t')n(t')dt' + \Sigma_i\lambda_i C_{i_0}\int_{t_0}^{t} e^{-\lambda_i(t'-t_0)}dt' - \Sigma_i\frac{\beta_i}{\ell}\int_{t_0}^{t} n(t')e^{-\lambda_i(t-t')}dt' \tag{4}$$

The equation for reactivity is $\frac{d\rho}{dt} = \alpha n(t)$; thus, $\rho(t) = \rho_0 + \alpha\int_{t_0}^{t} n(t')dt'$ (5).

Assume that $n(t) = n_0 e^{At}$ (6), and each time increment will start with $t_0 = 0$. Therefore, the time increment $\Delta t = t - t_0 = t = h$. Substituting the power equation (6) into reactivity equation (5) yields the reactivity equation used in the code.

$$\rho(t) = \rho_0 + \alpha\int_0^h n_0 e^{At'}dt' = \rho_0 + \frac{\alpha n_0}{A}\left(e^{Ah} - 1\right) \tag{7}$$

Substitute equation (6) into the precursor equation (3) and evaluate to derive the precursor equation used in the code.

$$C_i(t) = C_{i_0}e^{-\lambda_i(t-t_0)} + \frac{\beta_i}{\ell}\int_{t_0}^{t} n_0 e^{At'}e^{-\lambda_i(t-t')}dt'$$

$$= C_{i_0}e^{-\lambda_i h} + \frac{\beta_i n_0}{\ell}\int_0^h e^{-\lambda_i t}e^{(A+\lambda_i)t'}dt'$$

$$C_i(t) = C_{i_o}e^{-\lambda_i h} + \frac{\beta_i n_o e^{-\lambda_i h}}{\ell(A+\lambda_i)}\left(e^{(A+\lambda_i)h} - 1\right) \tag{8}$$

The power equation (4) can be evaluated by substituting in the reactivity equation (7) and the assumption equation (6).

$$n(t) = n_0 + \frac{1}{\ell}\int_0^h \left[\rho_0 + \frac{\alpha n_0}{A}(e^{At'} - 1)\right]n_0 e^{At'}dt' + \Sigma_i\lambda_i C_{i_0}\int_0^h e^{-\lambda_i t'}dt' - \Sigma_i\frac{\beta_i}{\ell}\int_0^h n_0 e^{At'}e^{-\lambda_i(t-t')}dt'$$

$$= n_0 + \frac{\rho_0 n_0}{\ell}\int_0^h e^{At'}dt' + \frac{\alpha n_0^2}{\ell A}\int_0^h e^{2At'}dt' - \frac{\alpha n_0^2}{\ell A}\int_0^h e^{At'}dt'$$

$$+ \Sigma_i\lambda_i C_{i_0}\left(-\frac{1}{\lambda_i}e^{-\lambda_i t'}\right)_0^h - \Sigma_i\frac{\beta_i n_0}{\ell}e^{-\lambda_i t}\int_0^h e^{(A+\lambda_i)t'}dt'$$

$$= n_0 + \frac{\rho_0 n_0}{\ell A}(e^{Ah} - 1) + \frac{\alpha n_0^2}{\ell A}\left(\frac{e^{2Ah} - 1}{2A} - \frac{e^{Ah} - 1}{A}\right) - \Sigma_i C_{i_0}\left(e^{-\lambda_i h} - 1\right) - \Sigma_i \frac{\beta_i n_0 e^{-\lambda_i h}}{\ell(A + \lambda_i)}\left(e^{(A+\lambda_i)h} - 1\right)$$

$$n(t) = n_o + \frac{\rho_0 n_0}{\ell A}\left(e^{Ah} - 1\right) + \frac{\alpha n_0^2}{\ell A^2}\left(\frac{1}{2}(e^{2Ah} + 1) - e^{Ah}\right) - \Sigma_i C_{i_0}(e^{-\lambda_i h} - 1) - \Sigma_i \frac{\beta_i n_0 e^{-\lambda_i h}}{\ell(A + \lambda_i)}\left(e^{(A+\lambda_i)h} - 1\right) \quad (9)$$

For the computer code to evaluate equation (9) must be solved for A. Thus, set (9) equal to (6).

$$0 = -n_0 e^{Ah} + n_0 + \frac{n_0 \rho_0}{\ell A}\left[e^{Ah} - 1\right] + \frac{\alpha n_0^2}{\ell A^2}\left(\frac{1}{2}\left(e^{2Ah} + 1\right) - e^{Ah}\right) - \Sigma_i C_{i_0}\left(e^{-\lambda_i h} - 1\right) - \Sigma_i \frac{\beta_i n_0 e^{-\lambda_i h}}{\ell(A + \lambda_i)}\left(e^{(A+\lambda_i)h} - 1\right)$$

The code uses a root finding routine to solve for A; however, a first approximation must be found. As an approximation let

$$e^{Ah} \approx 1 + Ah + \frac{A^2 h^2}{2}.$$

$$0 = n_0(1 - 1 - Ah) + \frac{n_0 \rho_0}{\ell A}(1 + Ah - 1) + \frac{\alpha n_0^2}{\ell A^2}\left(\frac{1}{2}\left(1 + 2Ah + \frac{4A^2 h^2}{2} + 1\right) - 1 - Ah - \frac{A^2 h^2}{2}\right)$$

$$- \Sigma_i C_{i_0}\left(e^{-\lambda_i h} - 1\right) - \Sigma_i \frac{\beta_i n_0 e^{-\lambda_i h}}{\ell(A + \lambda_i)}(1 + (A + \lambda_i)h - 1)$$

$$0 = -n_0 Ah + \frac{n_0 \rho_0 h}{\ell} + \frac{\alpha n_0^2}{\ell A^2}\left(\frac{A^2 h^2}{2}\right) - \Sigma_i C_{i_0}\left(e^{-\lambda_i h} - 1\right) - \Sigma_i \frac{\beta_i n_0 e^{-\lambda_i h} h}{\ell}$$

$$n_0 Ah = \frac{n_0 e_0 h}{\ell} + \frac{\alpha n_0^2 h^2}{2\ell} - \Sigma_i C_{i_o}\left(e^{-\lambda_i h} - 1\right) - \Sigma_i \frac{\beta_i n_o e^{-\lambda_i h} h}{\ell}$$

$$\text{Thus, } A = \frac{\rho_o}{\ell} + \frac{\alpha n_o h}{2\ell} - \Sigma_i \frac{C_{i_o}}{n_o h}\left(e^{-\lambda_i h} - 1\right) - \Sigma_i \frac{\beta_i e^{-\lambda_i h}}{\ell}$$

APPENDIX B. POINTRX PROGRAM

The program, POINTRX, is run in a DOS window. The name of the executable file is followed by the name of the input file, followed by the name of the output file. If the input file is not found in the same directory as the one being executed from and the directory is not indicated on the command line; then the program will abort with a message stating the input file was not found. If the output file is not identified on the command line, the program creates or appends a file called output. An example execution line will look like:

C:\pointrx aprf-in.txt aprf-out.txt

The input file does not need to be a wordpad, notepad, or any other word processor file, but it must be a text file. Any comments after a double slash (//) are ignored as comments, thus the sample input file, included in this appendix as page 2, has many comments to show the location of all the inputs. The output file will identify at the beginning of the file all the inputs used in the program, without the comments. After the program has read all the constants, the initial time will be the start time for the output. There must be at least two lines for the program to run so that the program has a start time and a stop time.

The code assumes that the power in each time increment follows the form: $n_o e^{Ah}$. The code determines the free parameter A (the inverse period) to satisfy the integral equations at the beginning and end of each time increment. To check whether the exponential form is correct throughout the time interval h, the code makes two comparisons. First, the code compares the power at the end of the interval using the previous value for A, with a new value for A, selecting the best value. Second, the code computes new parameter A by solving the integral equation over a time period h/2. If the two values do not match, the time interval is halved and the process repeated. For each time increment, an attempt is made to stretch the time period h.

The program uses the first time increment as the initial guess for the time variable, h. Thus, if a large time increment is used, the "bad guess" message will appear on the screen until the program finds an appropriate value. The program will run for any number of time intervals or reactivity insertions or withdrawals. The time increment is also the data output interval the program uses to write data to the output file; thus increase the time increment for less data and shorten it for more data. A sample beginning of an output file is included in this appendix as page 3.

```
//******DATA INPUT FILE FOR POINTRX PROGRAM -- APRF STD CORE VALUES******
//
//  fast fisson values from G.R. Keepin(c1965) using B of 0.0068
//    beta          lambda
      0.0002584   0.0127
      0.0014484   0.0317
      0.0012784   0.115
      0.0027676   0.311
      0.0008704   1.40
      0.0001768   3.87
//
//      thermal coefficients                neutron              inital
//degress C/kw-sec      cents/degree C    lifetime(sec)      core tempC)
      0.04683           -0.3              1.0e-8                25
//
//
//   time increment is the data output interval
//
//   initial time   time increment  initial reactivity  initial power
//       sec             sec              $                 kW
        0.0            1.e-5          1.05          0.001 // pulse rod insert
        1.e-3          1.e-4          0.00
        2.e-3          1.e-3          0.00
        50.0e-3        1.0e-3        -0.05                 // SB begins to move
        52.5e-3        1.0e-3        -0.10                 // scram
        55.0e-3        1.0e-3        -0.10                 // scram
        57.5e-3        1.0e-3        -0.25                 // scram
        60.e-3         1.0e-3        -0.5                  // scram
        70.e-3         1.0e-3        -1.0                  // scram
        80.e-3         1.0e-3        -1.0                  // scram
        90.e-3         1.0e-3        -1.0                  // scram
        100.e-3        1.0e-3        -1.0                  // scram
        120.e-3        1.0e-3        -1.0                  // scram
        130.e-3        1.0e-3        -1.0                  // scram
        140.e-3        1.0e-3        -1.0                  // scram
        150.e-3        1.0e-3        -1.0                  // scram
        160.e-3        1.0e-3        -1.0                  // scram
        170.e-3        1.0e-3        -1.0                  // scram
        180.e-3        1.0e-3        -1.0                  // scram
        190.e-3        1.0e-3        -1.0                  // scram
        200.e-3        1.0e-3        -1.0                  // scram
        210.e-3        1.0e-3        -1.0                  // SB completely out
        0.5            1.0e-2         0.00
        1.00           0.1            0.00
        10.0           1.0            0.00
        30.0
```

```
INPUT FILE DATA:
      0.0002584   0.0127
      0.0014484   0.0317
      0.0012784   0.115
      0.0027676   0.311
      0.0008704   1.40
      0.0001768   3.87
      0.04683           -0.3        1.0e-8                    25
         0.0            1.e-5           1.05          0.001
         1.e-3          1.e-3           0.00
         50.0e-3        1.0e-3         -0.05
         52.5e-3        1.0e-3         -0.10
         55.0e-3        1.0e-3         -0.10
         57.5e-3        1.0e-3         -0.25
         60.e-3         1.0e-3         -0.5
         70.e-3         1.0e-3         -1.0
         80.e-3         1.0e-3         -1.0
         90.e-3         1.0e-3         -1.0
         100.e-3        1.0e-3         -1.0
         120.e-3        1.0e-3         -1.0
         130.e-3        1.0e-3         -1.0
         140.e-3        1.0e-3         -1.0
         150.e-3        1.0e-3         -1.0
         160.e-3        1.0e-3         -1.0
         170.e-3        1.0e-3         -1.0
         180.e-3        1.0e-3         -1.0
         190.e-3        1.0e-3         -1.0
         200.e-3        1.0e-3         -1.0
         210.e-3        1.0e-3         -1.0
         0.5            1.0e-2          0.00
         1.00           0.1             0.00
         10.0           1.0             0.00
         30

BEGIN OUTPUT DATA:
     time        power (kW)      temp           rho         1/period       delta-t
 0.000000e+000 1.000000e-003 2.500000e+001 7.140000e-003
 1.000000e-005 9.503938e-003 2.500000e+001 7.140000e-003 1.054957e+005 2.242261e-008
 2.000000e-005 2.145182e-002 2.500000e+001 7.140000e-003 6.572550e+004 3.672489e-008
 3.000000e-005 3.823869e-002 2.500000e+001 7.140000e-003 5.178624e+004 4.905765e-008
 4.000000e-005 6.182468e-002 2.500000e+001 7.140000e-003 4.501189e+004 5.867678e-008
 5.000000e-005 9.496391e-002 2.500000e+001 7.140000e-003 4.115234e+004 9.312402e-008
 6.000000e-005 1.415261e-001 2.500000e+001 7.140000e-003 3.881214e+004 1.282834e-007
 7.000000e-005 2.069484e-001 2.500000e+001 7.140000e-003 3.729766e+004 1.203567e-007
 8.000000e-005 2.988707e-001 2.500000e+001 7.140000e-003 3.627950e+004 1.930491e-007
 9.000000e-005 4.280272e-001 2.500000e+001 7.140000e-003 3.559906e+004 1.649125e-007
 1.000000e-004 6.095005e-001 2.500000e+001 7.140000e-003 3.511921e+004 2.366062e-007
 1.100000e-004 8.644826e-001 2.500000e+001 7.140000e-003 3.479718e+004 2.246515e-007
 1.200000e-004 1.222749e+000 2.500000e+001 7.140000e-003 3.456455e+004 4.432042e-007
....
```

```
// POINTRX  - computes reactor power (n0) using weighted residual method


#include "stdafx.h"

#define LINELENGTH 121                                          // Max length of input/output
line
#define MAXTRIES 25                                             // Max number of times to
subdivide h

//global variables
//
static double smallNumber=1.0E-10;
double n0, rho, nRho, c[6], temp;
double alpha2, alpha1, ngentime, time_increment, time;
double power, nTime, nTime_increment, milestone;
double ttime, beta[6], lambda[6], beta_sum;

//*****************************************************************************
//  compute (exp(a*h)-1)/a safely, with no divide by zero.
//  f1 -> A
double aexp(double f1, double f2)
{
 if (fabs(f1) > smallNumber) return ((exp(f1*f2)-1.0)/f1);
 else return (f2 + 0.5*f1*f2*f2);
};

//*****************************************************************************
//  Put a line into output file.
//
void putLine(FILE *fp, char *line)
{
 int index, out;

 for (index=0; index<LINELENGTH; index++)
 {
        out = fputc(line[index], fp);
        if (line[index] == '\n') return;
 };
 return;
};


//*****************************************************************************
// Get a line from input file.
// Ignore lines after double-slash "//".
// Ignore blank lines
//
int getLine(FILE *fp, char *line)
{
 int index, ii;
 char letter;

 for (;;)                                                       // Read until line without //
 {
```

```
        memset(line, ' ', LINELENGTH);                          // Clear line

        for(index=0; index<LINELENGTH; index++)
        {
                letter = fgetc(fp);
                if (letter == EOF)                                      // Treat end-of-file as end-of-line
                {
                        letter = '\n';
                        if (index == 0)
                        {
                                printf("End of file found\n");
                                return 0;
                        };
                };
                line[index] = letter;
                if (letter == '\n')
                {                                                                       // Finished
reading line
                        for (ii=0; ii<index-1; ii++)
                        {
                                if ( (line[ii]=='/') && (line[ii+1]=='/') )     // Check for //
                                {
                                        line[ii] = '\n';
                                        break;
                                };
                        };
                        if (line[0] == '\n') break;                     // Check for empty line.
                        else return 1;                                                  // else done.
                };
        };
        if (index >= LINELENGTH)                                        // If an input line was too long
        {
                printf("Input line too long.");                 // inform
                line[LINELENGTH-1] = '\n';
                return 1;                                                               // and terminate
        };
 };
};

//*****************************************************************************************
int getData(FILE *fp)
{
 char line[LINELENGTH];

 printf("Retrieving data from file.\n");
 if (getLine(fp, line) == 0) return 0;                  // Get data from input file
 if (sscanf(line, "%Lg%Lg%Lg%Lg", &nTime, &nTime_increment, &nRho, &power) == 0) return 0; // read data
 nRho = beta_sum*nRho;                                                  // Convert reactivity
 if (power<1.e-6) power=1.0e-6;                                 // initial power > 0

 return 1;
}


//*****************************************************************************************
int initialization(FILE *fp1, FILE *fp2)
```

```
{
 int i;
 char line[LINELENGTH];

 printf("Initializing.\n");
 sprintf(line, "%s\n", "INPUT FILE DATA:");
 putLine(fp2, line);
 while(getLine(fp1, line) !=0)                                    // Write entire input file to output file
 {
        putLine(fp2, line);
 }
 fprintf(fp2, "\nBEGIN OUTPUT DATA:\n");
 fseek(fp1 ,0L, SEEK_SET);                                        // Reset input file pointer to beginning
of file
 for(i=0; i<6; i++)
 {
        if (getLine(fp1, line) == 0) return 0;                    // Get data from input file
        if (sscanf(line, "%Lg%Lg", &beta[i], &lambda[i]) == 0) return 0; // read data
 }
 if (getLine(fp1, line) == 0) return 0;                           // Get data from input file
 if (sscanf(line, "%Lg%Lg%Lg%Lg", &alpha1, &alpha2, &ngentime, &temp) == 0) return 0; // read data
 for(beta_sum=0, i=0; i<6;i++) beta_sum += beta[i];
 alpha2 = alpha2*beta_sum*alpha1/100;                             //converttemp coeff cents/degree C to reactivity/kw-sec
 return 1;
}

//*****************************************************************************

double function(double Aest, double h)
{
 int i;
 double sum=0.0;

 sum = n0-n0*exp(Aest*h);
 sum = sum + n0*rho*aexp(Aest,h)/ngentime;

 if(fabs(Aest)<0.00001)
        sum = sum + (alpha2*n0*n0*h*h)/(ngentime*2);
 else
        sum = sum + (alpha2*n0*n0/(ngentime*Aest*Aest))*(0.5*(exp(2*Aest*h)+1)-exp(Aest*h));

 for(i=0;i<6;i++)
 {
        sum = sum - c[i]*(exp(-lambda[i]*h)-1.);
        sum = sum - (beta[i] * n0 * exp(-lambda[i]*h) * aexp(Aest+lambda[i],h))/ngentime;
 }
 return(sum);
}

//************************************************
// Finds a value x which gives func(x, h)=0.
// Input a first approximation x.
double root_find(const double x, const double h)
{
 double x_low, x_hi;                                              // These bracket x
 double f_low, f_hi, f_x;                                         // func(x_low), func(x_hi), func(x)
```

```
 double x_low_old, x_hi_old, f_low_old, f_hi_old;
 double x_try, f_try, x_try1, f_try1, offset;
 double convergeS=1.0E-20;                          // Done if abs(func(x))<converge
 double convergeX=1.0E-5;                           // Done if x_low and x_hi match
 double eps=1.0e-20;                                // Small number
 double perturb=0.2;                                // Amount to perturb x in
initial search
 double limit1=1000.0,limit2=1.0E+10;               // Limits range of search
 double P, Q, R;
 int iter, max_iters=100;                           // Limit iterative improvement.

 f_x = function(x, h);
 if (fabs(f_x)<eps) return x;

 // Need to bracket x. Want to end with func(x_low)*func(x_hi)<0.
 // This sign change means that desired x is between x_low and x_hi.
 // Vary x and look for a crossing of axis.
 // Expand region around x until zero crossing found.
 offset = fabs(x)+0.1;                              // scale factor to search around x
 x_low_old = x;                                     // Lower limit of last search.
 x_hi_old  = x;                                     // Upper limit of last search
 f_low_old = f_x;
 f_hi_old  = f_x;
 do
 {
        x_low = x_low_old - perturb*offset;         // Vary low.
        f_low = function(x_low, h);
        if (f_low*f_x < 0)
        {
                x_hi = x_low_old;                   // Found zero crossing
                f_hi = f_low_old;
        }
        else                                        // Did not cross
zero
        {
                x_low_old = x_low;                  // March down if needed
                f_low_old = f_low;

                x_hi = x_hi_old + perturb*offset;   // vary high
                f_hi = function(x_hi, h);
                if (f_hi*f_x < 0)
                {                                   // Found
zero crossing on high side
                        x_low = x_hi_old;
                        f_low = f_hi_old;
                }
                else
                {
                        x_hi_old = x_hi;            // No crossing. Extend boundary.
                        f_hi_old = f_hi;
                };
        };
        perturb = 1.2*perturb;                      // Increase range for next search
        if (perturb>limit1) printf("%s"," Poor guess.\n");
        if (perturb>limit2)
        {
```

```
                printf("%s"," Search for root stopped.\n");
                return x;
        };
} while (f_low*f_hi > 0);                                       // Check to see if root bounded

// Root is bounded by x_low and x_hi.
// Try to improve root by selecting a new value x in the middle and fitting a quadratic.
// Use end best points to develop new x_low and x_hi.
for (iter=0; iter<max_iters; iter++)
{
        x_try = 0.5*(x_low+x_hi);                               // Point in the middle
        f_try = function(x_try, h);                             // Evaluate function there.
        if (fabs(f_try)<convergeS) return x_try;        // Got lucky. Done.
        if ( fabs(f_low-f_try)<eps || fabs(f_hi-f_try)<eps ||
                fabs(f_low-f_hi)<eps )                                  // divide by 0?
        {                                                                               // Use
existing value;
                x_try1 = x_try;
                f_try1 = f_try;
        }
        else
        {                                                                               // Find
quadratic guess
                P = f_low-f_try;
                Q = f_low-f_hi;
                R = f_try-f_hi;
                x_try1 = f_try*f_hi*x_low/(P*Q);
                x_try1 -= ((f_low*f_hi*x_try)/(P*R));
                x_try1 += ((f_low*f_try*x_hi)/(Q*R));
                f_try1 = function(x_try1, h);
        };
        if (f_try*f_low < 0)                                    // At least divide region by 2
        {
                x_hi = x_try;
                f_hi = f_try;
        }
        else
        {
                x_low = x_try;
                f_low = f_try;
        };
        if ( (x_try1>x_low) && (x_try1<x_hi) )
        {                                                                               // x_try1
within boundary
                if (fabs(f_try1)<convergeS) return x_try1;  // Done.
                if (f_try1*f_low < 0)
                {
                        x_hi = x_try1;
                        f_hi = f_try1;
                }
                else
                {
                        x_low = x_try1;
                        f_low = f_try1;
                };
        };
```

```
        if (fabs(f_hi-f_low)<convergeS) return (0.5*(x_hi+x_low));    // Return if both are close to 0
        if (fabs(x_hi-x_low)<convergeX) return (0.5*(x_hi+x_low));  // Return if x_hi and x_low match
 };
 printf("%s"," Exceeded iteration limit in root_find.\n");
 return (0.5*(x_hi+x_low));
};

//*****************************************************************************************
// Can use either calculated vaule or old value.
//
double calculate_A(double h)
{
 int i;
 static double Aold = 0.0;
 double Aguess=0.0;

 for(i=0; i<6; i++)
 {
        Aguess =Aguess-(c[i]*aexp(h,-lambda[i])/n0)-(beta[i]*exp(-lambda[i]*h)/ngentime);
 }
 Aguess = Aguess + rho/ngentime+alpha2*n0*h/(2*ngentime);
 if (fabs(function(Aold, h)) < fabs(function(Aguess, h))) Aguess = Aold;
 Aold = root_find(Aguess, h);
 return Aold;
};

//*****************************************************************************************
// Compute next time at which to print data
//
void nextMilestone()
{
 if ((milestone + time_increment)> nTime)
        milestone = nTime;
 else milestone = (milestone + time_increment);
};

//*****************************************************************************************
int main(int argc, char *in[])
{
  FILE *in_file;
 FILE *out_file;
 double deltaTime, h, Afactor;
 double Atry, Ahalf;
  int ii, i;
 double tolerance = 1.0E-6;
 char line[LINELENGTH];

  in_file = fopen(in[1], "r");                                    // Open input file
 if (in_file == NULL)
 {
        printf("Cannot open %s for input.\n", in[1]);
        return 1;
 };
 if(in[2]==NULL) out_file = fopen("output.txt", "a+");
 else out_file = fopen(in[2], "a+");                              //append output file, create if needed
 if (out_file == NULL)
```

```
{
        printf("Cannot open file for output.\n");
        return 1;
};
if (initialization(in_file, out_file)==0) return 1;        // initialize constants

sprintf(line," %s\n","  time     power (kW)   temp      rho      1/period     delta-t");
putLine(out_file, line);                                                    // milestone => print;

if (getData(in_file)==0) return 1;                                  // get starting data
n0 = power;
time = nTime;
time_increment = nTime_increment;
rho = nRho;
for(ii=0; ii<6; ii++) c[ii]=beta[ii]*n0/(lambda[ii]*ngentime); // initialize precursors
if (getData(in_file)==0) return 1;                                  // get next time marker
sprintf(line," %e %e %e %e \n", time, n0, temp, rho);
putLine(out_file, line);

h = time_increment;                                                          // Initial guess for h
nextMilestone();

while(time_increment>0.0)                                                    // Use this as end of problem marker
{
        Atry = calculate_A(h);
        for(ii=0; ii<MAXTRIES; ii++)
        {
                Ahalf = calculate_A(h/2.);                          // Keep subdividing h until A stabilizes
                if ((fabs(Atry-Ahalf)*h)<tolerance) break;
                Atry = Ahalf;
                h = h/2.0;
        };
        if (ii==MAXTRIES) printf("At time = %LE A did not converge",time);
        if ((time+h+smallNumber)>= milestone) deltaTime = milestone-time;
          else deltaTime = h;
        time = time + deltaTime;
        n0 =  n0*exp(Atry*deltaTime);
        if(n0>0.5)                                                                  // No temperature
adjustments below 500watts
        {
                Afactor = aexp(Atry,deltaTime);
                temp = temp + alpha1*n0*Afactor;
                rho = rho + alpha2*n0*Afactor;
        }
        for (i=0; i<6; i++)
        {
                c[i] = c[i]*exp(-lambda[i]*deltaTime) +
                        ((beta[i]*n0*exp(-lambda[i]*deltaTime)/ngentime)*aexp(Atry+lambda[i],deltaTime));
        };
        if (time >=(milestone-smallNumber))
        {
                sprintf(line," %e %e %e %e %e %e\n", time, n0, temp, rho, Atry, h);
                putLine(out_file, line);                                    // milestone => print;
                nextMilestone();
                if (time >= (nTime-smallNumber))
                {
```

```
                    time_increment = nTime_increment;
                    rho = rho + nRho;
                    if ( (nRho > beta_sum/1000.) && (h>1.0e-6) ) h = 1.0e-6;
                    if (getData(in_file)==0) return 1;          // get next time marker
                    nextMilestone();
                }
        };
        h = h*5.0 + 1.0e-9;                                     // Try to stretch h
};

  fclose(in_file);                                              // Close input/output files
 fclose(out_file);
  printf("Finished.\n");
 return 0;
}
```


(Page B-12 Blank)

APPENDIX C. HEAT CAPACITY DERIVATION

Figure 1 is a plot of the temperature data collected from thermocouple No. 7 during an 8-kW steady-state operation at 20 feet. No cooling was used during the run so that the temperature rise should be similar to that of a pulse operation. However, the graph is not exactly linear; as expected, the higher the temperature the more heat will be radiated away from the core. This is more evident in Figure 2, which is a plot of the heat capacity obtained from the data from the same operation. Low power steady-state operations will eventually reach an equilibrium temperature where the heat generated is equal to the heat being radiated away. Time zero in the graph is the point where the reactor is at 1/e for 8 kW, and reactor power is at 8 kW at 120 seconds.

To determine the heat capacity of the core, find the slope of the line by dividing a change in temperature by the change in time:

$$\frac{\Delta T}{\Delta t} = \frac{219 - 99}{540 - 180} = \frac{120}{360} = 0.3333\ ^{\circ}C/sec$$

Next, divide this number by the operating power level to get 0.04167 °C/kW-sec. This is within 5 percent of the value derived from previous operating data on the following sheets. The shutdown time for this operation was 600 seconds. Beyond this point the core is cooling down, with no power input, by radiating heat. The slope of this portion of the curve is -0.14 °C/sec. If this is added to the heat capacity, it raises the value to 0.0590 °C/kW-sec, and there is only a 5 percent difference between the heat capacity derived by this method and the method using pulse and temperature data given in the main report. Furthermore, this value compares to the peak of the heat capacity from Figure 2, 0.054 °C/kW-sec.

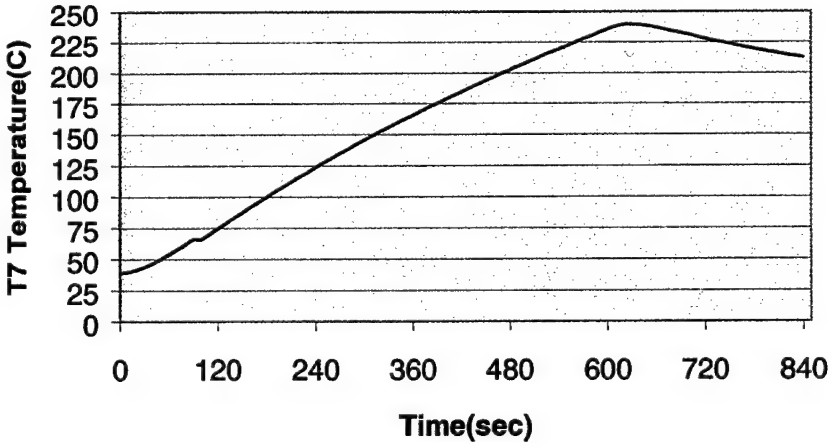


Figure 1. 8KW - no cooling - SS02-60.

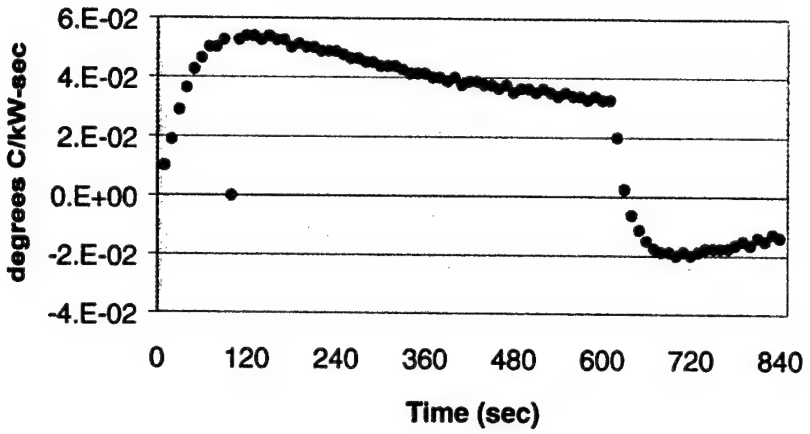


Figure 2. Heat capacity.

Changing the heat capacity does not alter the reactor period, pulse width, or temperature change, only the integrated power. That is why this parameter was used to calibrate the program to fit empirical data. Furthermore, since temperature changes are not affected by the heat capacity, the heat capacity used for temperature analysis is irrelevant.

FIGURE 31

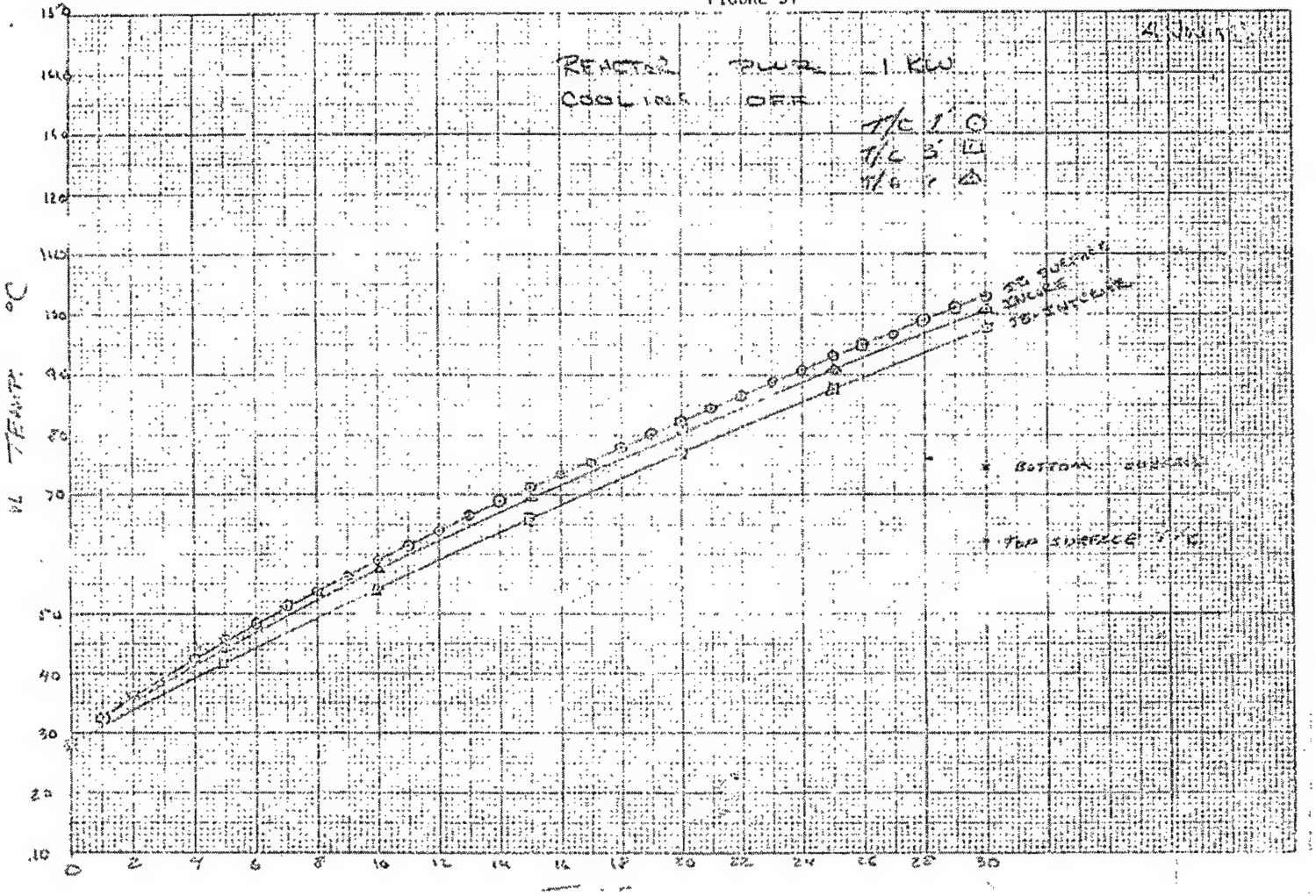
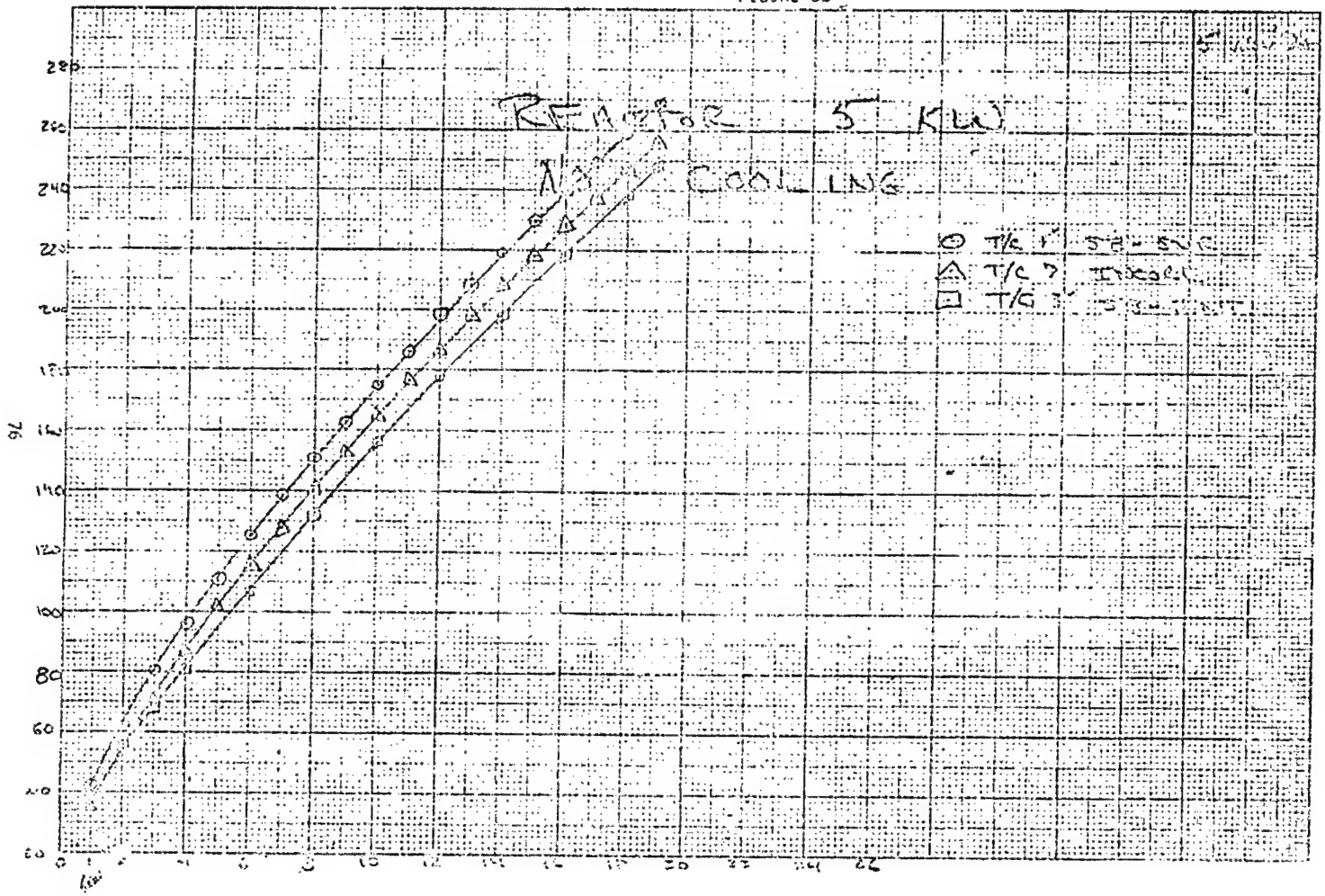


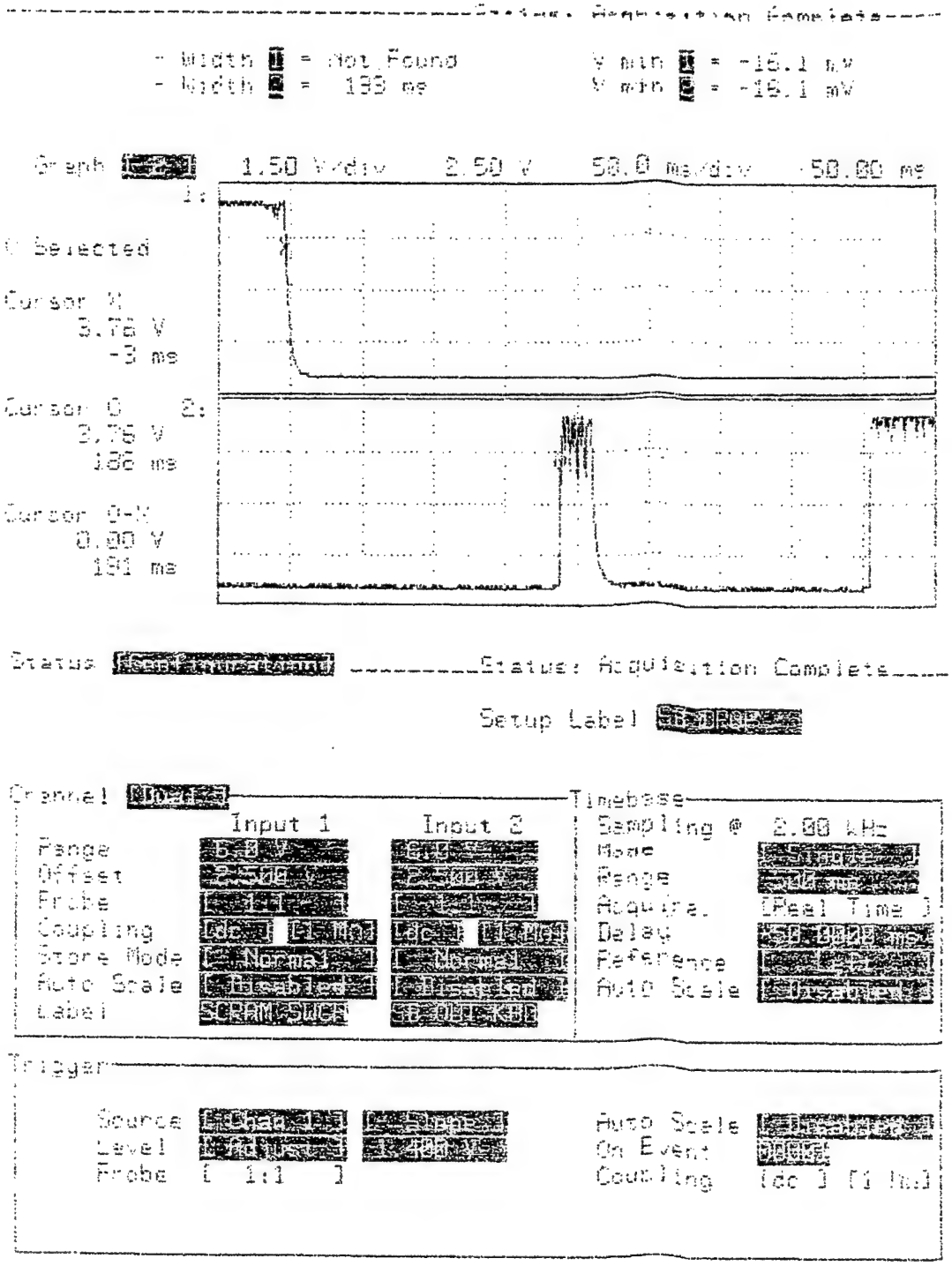
FIGURE 33

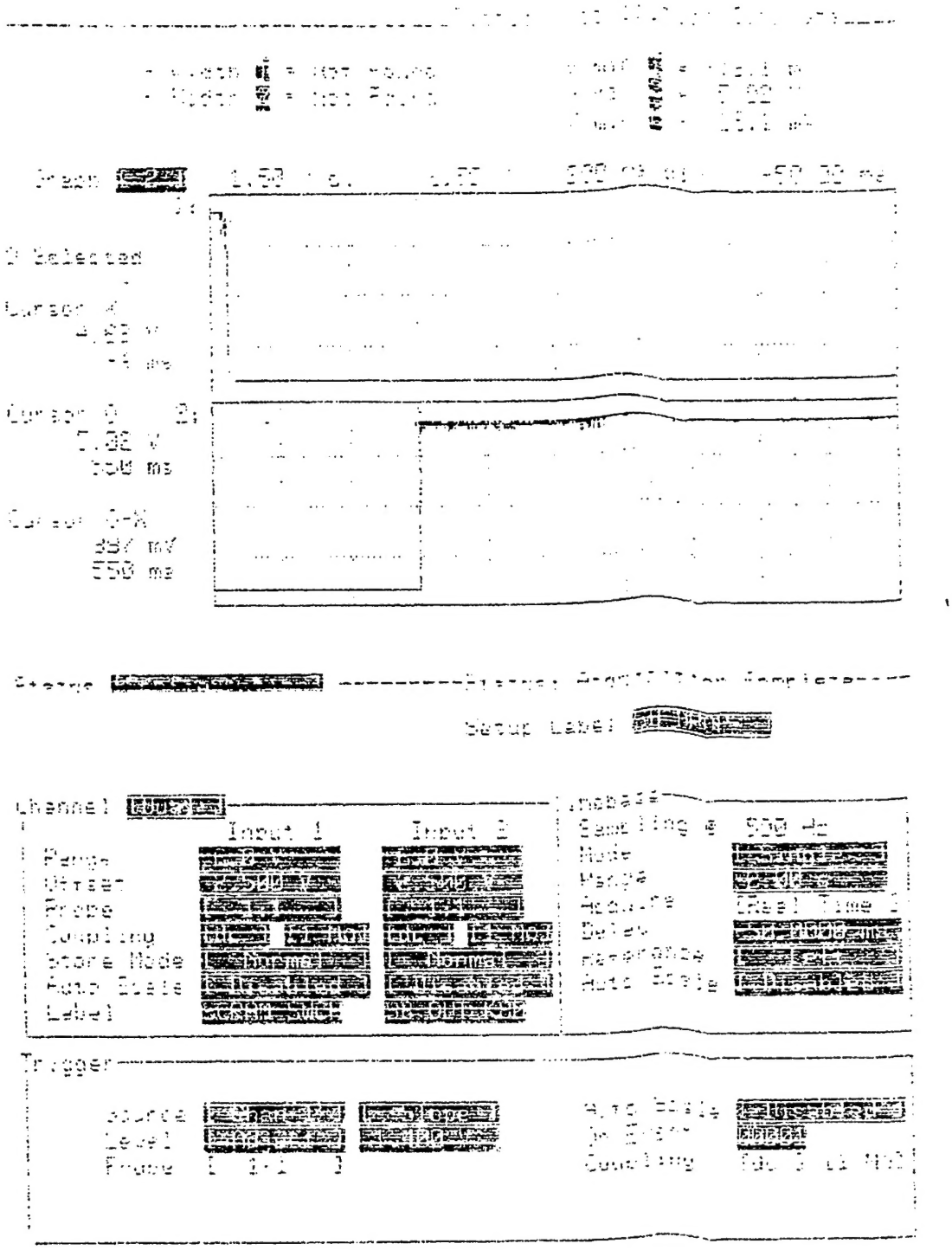


(Page C-6 Blank)

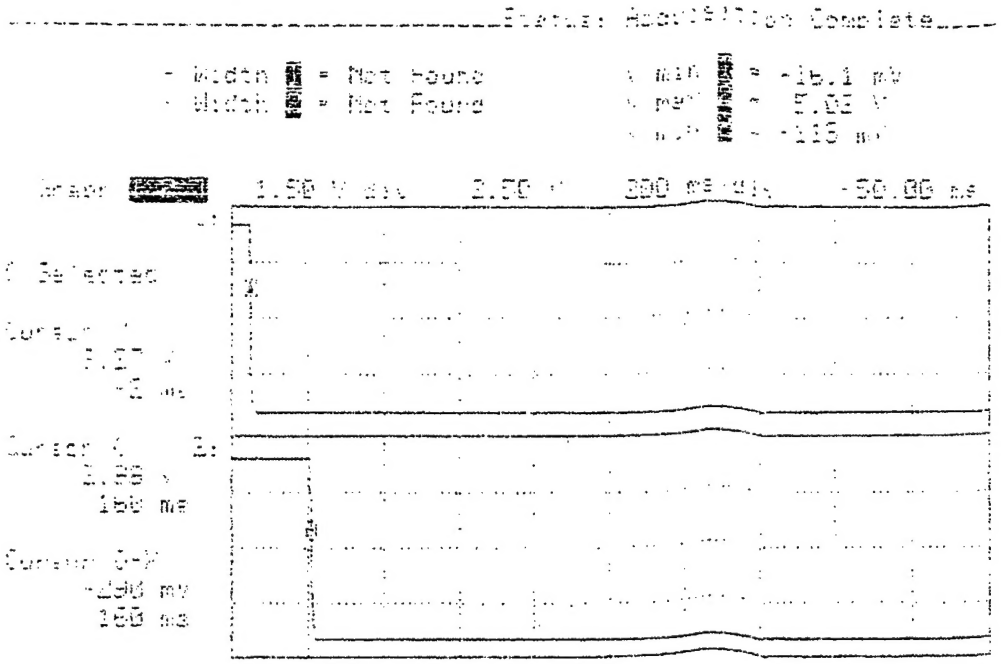
APPENDIX D. SAFETY BLOCK AND PULSE ROD TIMING

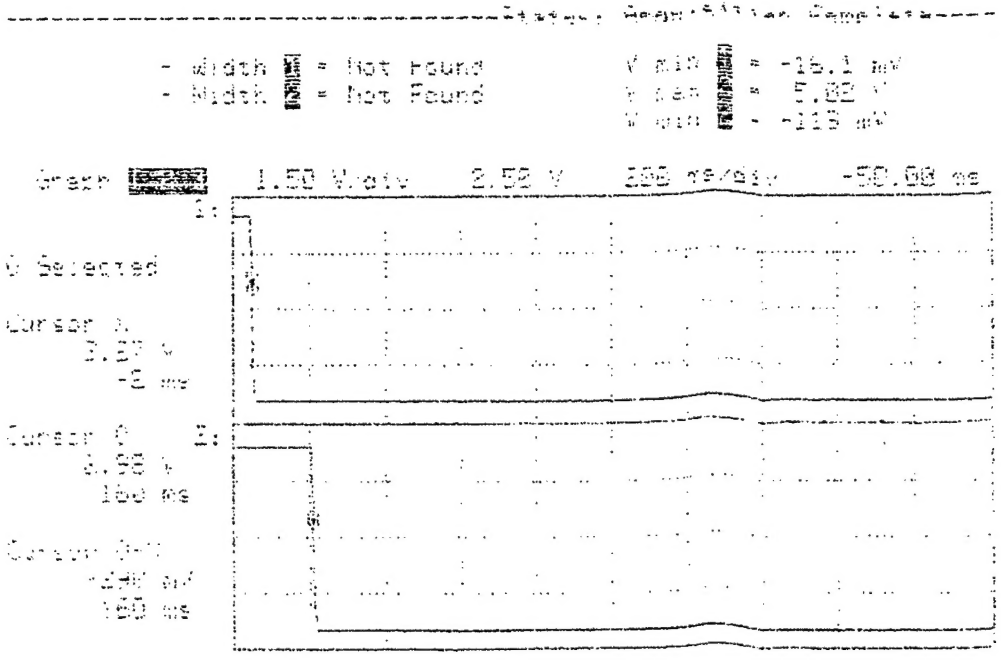
Safety Block Drop Test June 2002



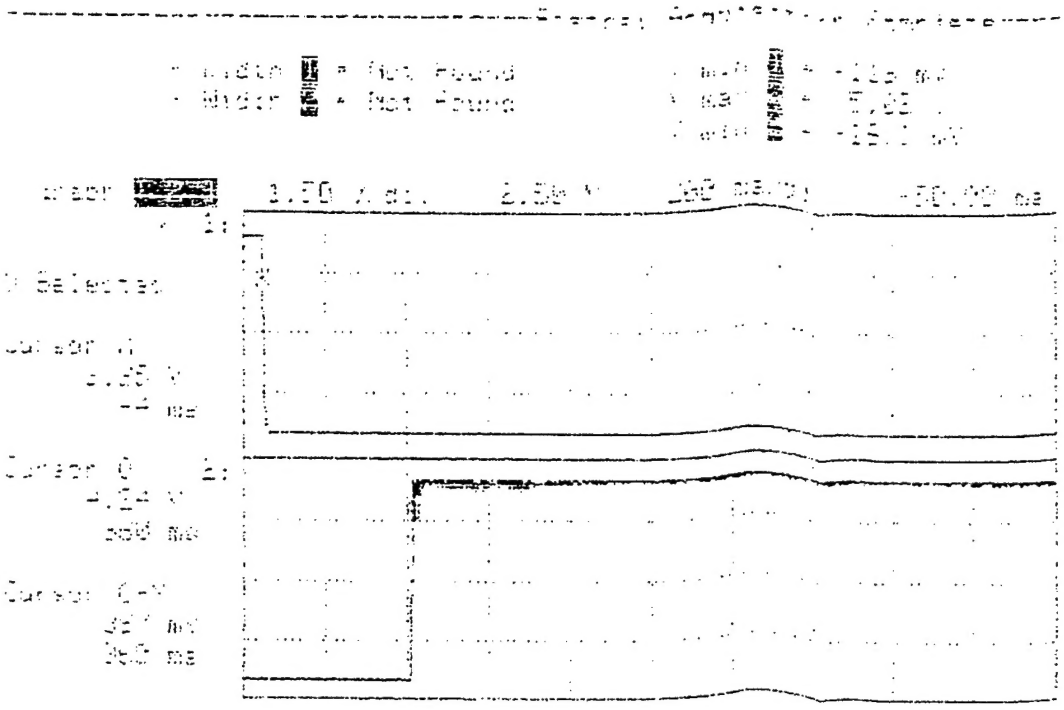


Scram signal to pulse rod out

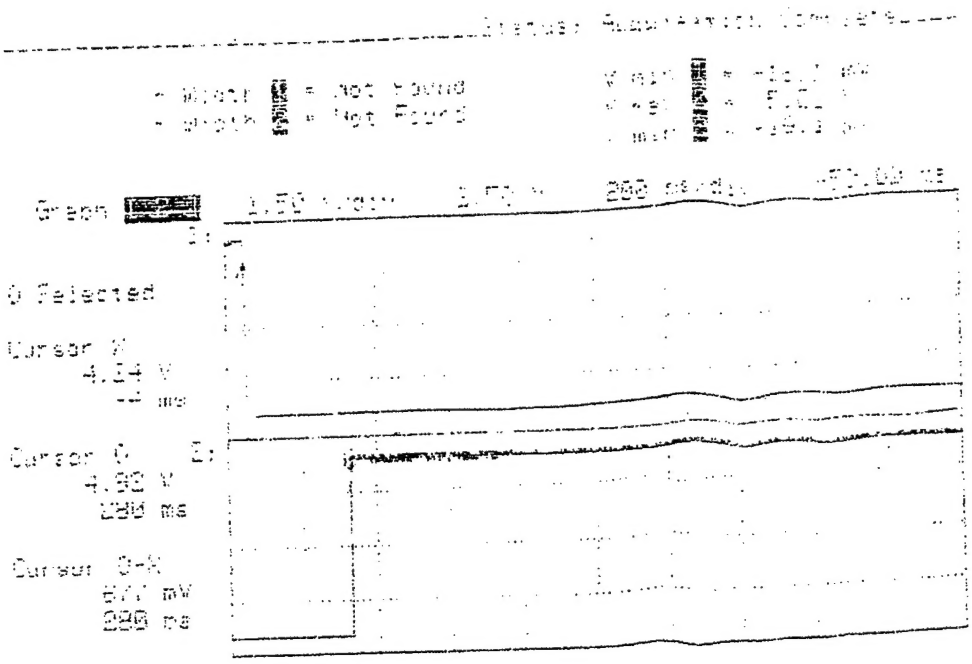


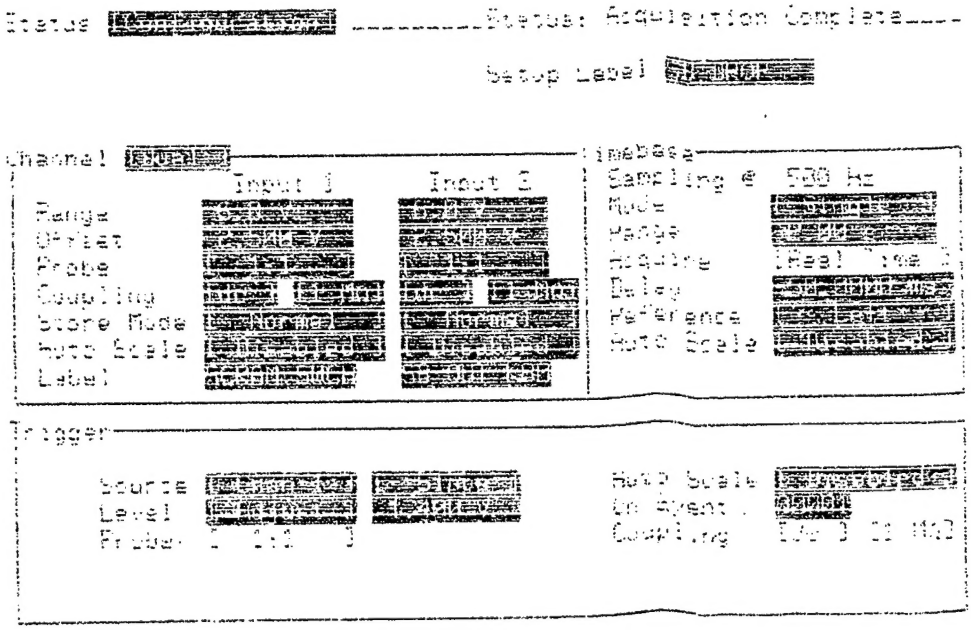


Scram signal to pulse rod start out



Pulse rod start out to pulse rod out





Pulse rod start out to pulse rod out using withdraw

(Page D-6 Blank)